IN THE DRAWINGS:

The attached sheet of drawings includes changes to Fig. 3. This sheet, which includes Figs. 2-3, replaces the original sheet including Figs. 2-3.

Attachment: Replacement Sheet

Annotated Sheet Showing Changes

REMARKS

This is in response to the Notice of Non-Responsive Response and is intended as a full and complete response to the Office Action dated May 21, 2004, having a shortened statutory period for response set to expire on August 21, 2004. Please reconsider the claims pending in the application for reasons discussed below.

In the specification, the paragraphs 20, 21, 37, 42, and 48 have been amended to correct minor editorial problems. New paragraph 19.1 is added to include a brief description of Figure 9A. Claims 1-28 are pending in the application. Fig. 3 has been amended. Claims 1-29 remain pending following entry of this response. Applicants submit that the amendments do not introduce new matter.

Claims 1-28 stand rejected under 35 U.S.C. § 102(e) as being anticipated by *Ungar* (US 6,085,035). Respectfully, applicants traverse the objection. *Ungar* and Applicants' invention do both involve complier optimization techniques; however, *Ungar* does not teach what is presently claimed. To understand the differences a brief discussion about compiler functioning is warranted.

A compiler is a computer program that translates a computer program written in one computer language (called the source language) into a program written in another computer language (called the output or the target language). Most compilers translate source code written in a high level language to object code or machine language that may be directly executed by a computer or a virtual machine. Typical compilers output objects containing machine code augmented by information about the name and location of entry points and external calls (to procedures not contained in the object). A set of object files, may then be linked together to create the final executable which can be run directly by a user.

"Call linkage" describes the process of tying a procedure called within one module call to the, possibly different, module that contains the definition for the procedure being called. Additionally, many procedures include arguments that must be passed from the called procedure from the calling procedure location. For example, a procedure to add to variables and return can be defined as follows:

```
int add (int x, int y)
{       return (x+y);          }
```

When the "add" procedure is called from one module, program control must shift to the location of the "add" procedure. Additionally, the program must pass (i.e., must make available) the parameters (variables) "x" and "y" to the "add" procedure.

In programming languages, a variable can be thought of as a place to store a value in computer memory. When a program declares a variable, the compiler typically sets aside a space in memory to store the value given to that variable. In a statically-typed language, the storage space is dependent on the data type of the declared variable (e.g. a C compiler will typically allocate 1 byte of storage to a character variable and 4 bytes of storage to an integer variable) Later, when the variable is no longer in use, this space can be reclaimed. A variable's name is frequently used as an identifier, and programmers typically name variables to be descriptive of the data they hold. For example, a variable in C might be called height or numberOfCats or account_balance.

Some programming languages allow for the data type (e.g., character, pointer, integer, real) associated with a variable to change over the course of program execution. That is, different data types at different points during the execution of a computer program may be stored in the same variable. In addition, pointer variables (addresses in memory) may point to different data types. Accordingly, the complier cannot, at compile time, assign a fixed amount of memory to a variable that may hold multiple data types over the course of program execution.

Ungar discloses a method for optimizing the object code generated by a compiler using the data types associated with variables. In particular, the optimization techniques of *Ungar* describes variable types as "mutable" and "immutable" to define meta-information about a particular variable to optimize object code.

Applicants claim a method for optimizing call-linkage for each call to a procedure that occurs in a program, regardless of whether a variable is "mutable" or "immutable." In contrast, as described below, *Ungar* discloses a complier optimization technique using the data types associated with a computer program's variables. Put simply, although both Applicants' claims and *Ungar* may be used to optimize object code

generated from source modules, they do so in different ways, using different information, to achieve different optimization results.

Regarding claims 1-13: Applicants claim "a method for optimizing a run time for an object code generated from a source code" that includes the steps of (i) extracting information for each procedure call contained in the source code, (ii) selecting a call linkage between a caller procedure and a callee procedure for each procedure call using the extracted information, where the selected call linkage is optimized to minimize a run time of an object code generated from the source code, (iii) generating the object code from the source code, and (iv) running the object code using the selected call linkages for each procedure call. Admittedly, both *Ungar* and Applicants' claims are directed to optimization methods to optimize object generated from source code, and both include generating the object code from the source code (i.e., both contemplate actually compiling the source code to produce an executable computer program). Beyond this, however, the similarities end.

As stated, Applicants claim an optimization technique that includes (i) extracting information for each procedure call contained in the source code, (ii) selecting a call linkage between a caller procedure and a called procedure for each procedure call using the extracted information, where the selected call linkage is optimized to minimize a run time of an object code generated from the source code. *Ungar* teaches a method for optimizing object code based on variable "types." *Ungar* Col. 3 l. 30-35. As used by *Ungar*, a variable may be "immutable" or "mutable" depending on whether a variable can (or actually does) store different data types. *Ungar* Col. 8 l. 11-52, Figure 3. Using a dynamic compilation environment (i.e., the compiler runs concurrent with program execution) the method taught by *Ungar* monitors the executing program to determine actual, or likely variable types and usage patterns, and creates compiled object code accordingly. *Ungar* Col. 8 l. 35-40. *Ungar* also teaches that in a non-dynamic environment "using a static compiler, the program is first profiled and the optimization process is invoked during a subsequent compilation that utilizes the profiled data." *Ibid.* In either case, the methods disclosed by *Ungar* require data regarding program execution. This data may be obtained from an executing program in a dynamic

compilation environment, or from a monitoring process that monitors the actively of an executable program compiled in a static compilation environment.

The present claims recite optimizing source code by selecting on one of multiple call linkage options determined using information extracted from the source code for the program being compiled contained in multiple modules. In contrast, *Ungar* discloses optimizing executable code based on profiling data obtained either from an executing program used for a *subsequent* compilation (and optimization) of the same source program or from monitoring an executing program and compiling subsequent portions of the program based on the monitored activity. For example, the *Ungar* discloses: "A first preferred embodiment optimizes both a called routine and the call site dependent on the types of data values passed from the call site to the called routine. ...The invention detects variables that have immutable types (from the 'determine type usage pattern' *305*)." *Ungar*, Col. 8 l. 52-67. The "determine type usage pattern" disclosed by *Ungar* requires the compiler to determine executing program behavior to optimize the executable code. "This determination can be made from run-time data gathered by a profiler-instrumented program compiled by a static compiler or by a [sic] the run-time and compiler states of a program compiled by a dynamic compiler." *Ungar*, Col. 8 l. 35-40.

Applicants, however, claim an optimization technique that relies on selecting a call linkage between a caller procedure and a callee procedure for each procedure call using the extracted information, where the selected call linkage is optimized to minimize a run time of an object code generated from the source code. The extracted information is derived from the various source modules being compiled by the complier not from the "type usage patterns" taught by *Ungar*. Applicants' claim selecting a call linkage based only on the information extracted from the source code, without any reliance (or even awareness) of later program execution, whereas *Ungar* discloses an optimization technique that requires some reliance on executing program behavior, *Ungar*, Col. 8 l. 35-40.

Accordingly, Ungar fails to teach show or suggest a method for optimizing a run time for an object code generated from a source code that includes the steps of extracting information for each procedure call contained in the source code, selecting a call linkage between a caller procedure and a callee procedure for each procedure call using the extracted information, where the selected call linkage is optimized to minimize the run time of the object code generated from the source code, and generating the object code from the source code; and running the object code using the selected call linkages for each procedure call.

Regarding claims 14-28: Claims 14-20 claim an apparatus for optimizing a run time of an object code generated from a source code, and claims 21-28 claim an a computer readable medium storing a software program that performs the operations of Applicants' invention. The Examiner rejects these claims as "all claim limitations also have been addressed and/or covered in the cited areas as set forth above," and the rejection of these claims does not assert any independent basis to support the rejection. Accordingly, the traversal to the rejection of claims 1-13 as set forth above applies with equal force to claims 14-28.

The secondary references made of record are noted. However, it is believed that the secondary references are no more pertinent to the Applicants' disclosure than the primary references cited in the office action. Therefore, Applicants believe that a detailed discussion of the secondary references is not necessary for a full and complete response to this office action.

Having addressed all issues set out in the office action, Applicants respectfully submit that the claims are in condition for allowance and respectfully request that the claims be allowed.

Respectfully submitted,

Gero G. McClellan
Registration No. 44,227
MOSER, PATTERSON & SHERIDAN, L.L.P.
3040 Post Oak Blvd. Suite 1500
Houston, TX 77056
Telephone: (713) 623-4844
Facsimile: (713) 623-4846
Attorney for Applicant(s)

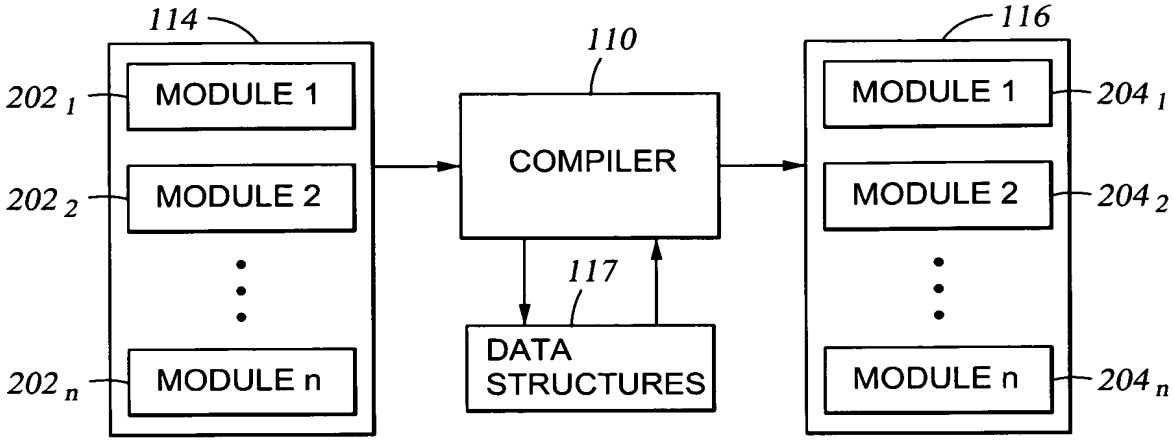

Fig. 2

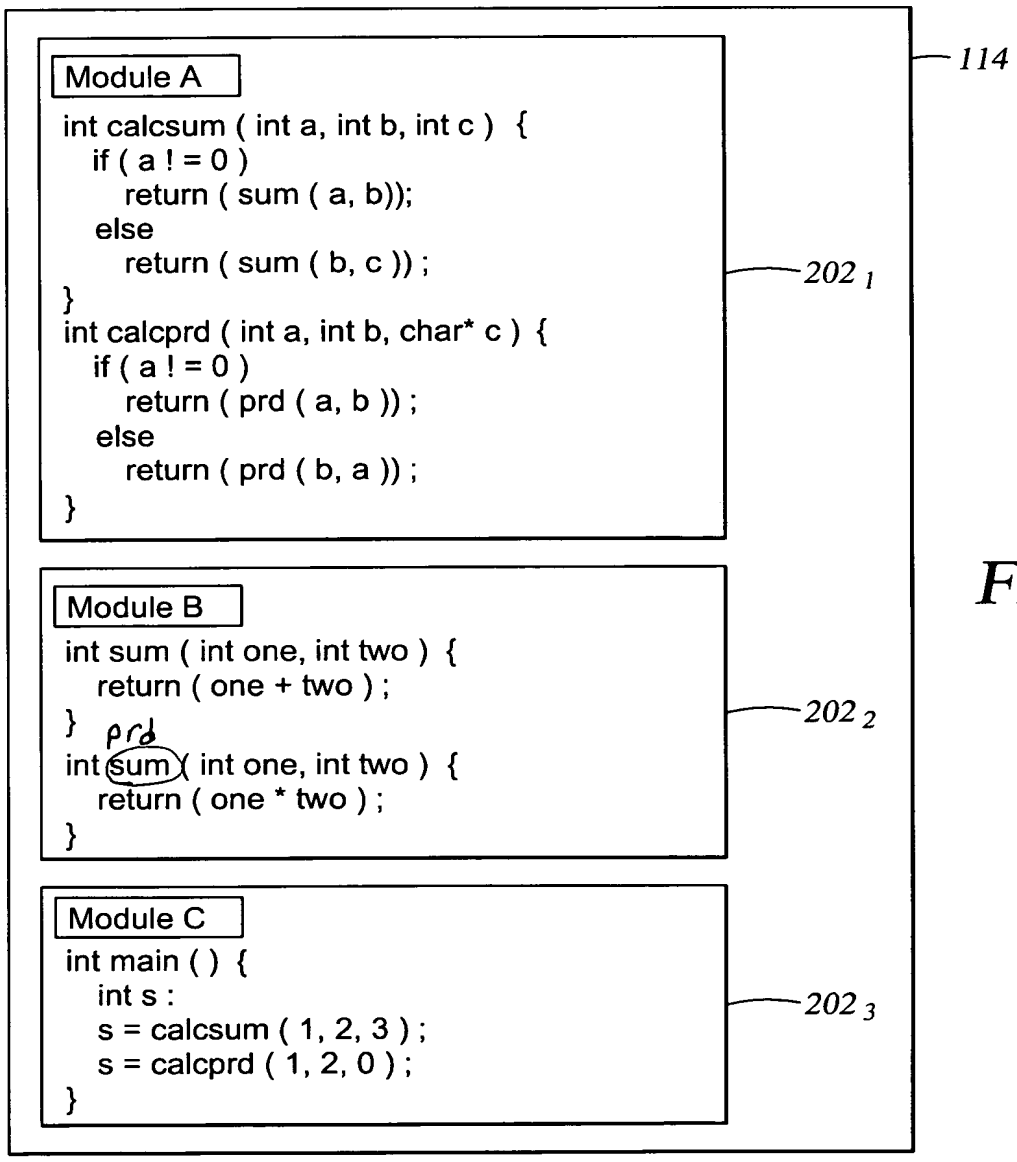

Fig. 3